Logical Methods in Computer Science
Vol. 7 (1:8) 2011, pp. 1–24
www.lmcs-online.org

Submitted Jun. 11, 2010
Published Mar. 23, 2011

FROM COINDUCTIVE PROOFS TO EXACT REAL ARITHMETIC: THEORY AND APPLICATIONS

ULRICH BERGER

Swansea University, Swansea SA2 8PP, Wales, UK
e-mail address: u.berger@swansea.ac.uk

Abstract. Based on a new coinductive characterization of continuous functions we extract certified programs for exact real number computation from constructive proofs. The extracted programs construct and combine exact real number algorithms with respect to the binary signed digit representation of real numbers. The data type corresponding to the coinductive definition of continuous functions consists of finitely branching non-wellfounded trees describing when the algorithm writes and reads digits. We discuss several examples including the extraction of programs for polynomials up to degree two and the definite integral of continuous maps. This is a revised and substantially extended version of the conference paper [6].

1. Introduction

Most of the recent work on exact real number computation describes algorithms for functions on certain exact representations of the reals (for example streams of signed digits [18, 19] or linear fractional transformations [17]) and proves their correctness using a certain proof method (for example coinduction [16, 11, 8, 30]). Our work has a similar aim, and builds on the work cited above, but there are two important differences. The first is *methodological*: we do not 'guess' an algorithm and then verify it, instead we *extract* it from a proof, by some (once and for all) proven correct method. That this is possible in principle is well-known. Here we want to make the case that it is also feasible, and that interesting and nontrivial new algorithms can be obtained (see also [33, 9] for related work on program extraction in constructive analysis and inductive definitions). The second difference is *algorithmic*: our method represents a uniformly continuous real function not by a *function* operating on representations of reals, but by an infinite *tree* that contains information not only about the real function as a point map, but also about its modulus of continuity. Since the representing tree is a pure data structure (without function component) a lazy programming language, like Haskell, will memoize computations which improves performance in certain situations.

A crucial ingredient in the proofs (that we use for program extraction) is a coinductive definition of the notion of uniform continuity (u. c.). Although, classically, continuity and uniform continuity are equivalent for functions defined on a compact interval (we only

1998 ACM Subject Classification: ??

Key words and phrases: Proof theory, realizability, program extraction, coinduction, exact real number computation.

DOI:10.2168/LMCS-7 (1:8) 2011

consider such functions), it is a suitable constructive definition of *uniform* continuity which matters for our purpose. For convenience, we consider as domain and range of our functions only the interval $\mathbb{I} := [-1, 1] = \{x \in \mathbb{R} \mid |x| \leq 1\}$ and, for the purpose of this introduction, only unary functions. However, later we will also look at functions of several variables where one has to deal with the non-trivial problem of choosing the input streams from which the next digit is consumed, a choice which can have a big influence on the performance of the program.

We let $\mathrm{SD} := \{-1, 0, 1\}$ be the set of (binary) *signed digits*. By SDS we denote the set of all infinite streams $a = a_0 : a_1 : a_2 : \ldots$ of signed digits $a_i \in \mathrm{SD}$. A signed digit stream $a \in \mathrm{SDS}$ represents the real number

$$\sigma(a) := \sum_{i \geq 0} a_i 2^{-(i+1)} \in \mathbb{I}$$

A function $f : \mathbb{I} \to \mathbb{I}$ is *represented* by a stream transformer $\hat{f} : \mathrm{SDS} \to \mathrm{SDS}$ if $f \circ \sigma = \sigma \circ \hat{f}$. The coinductive definition of uniform continuity, given in Sect. 3, allows us to extract from a constructive proof of the u. c. of a function $f : \mathbb{I} \to \mathbb{I}$ an algorithm for a stream transformer $\hat{f}$ representing f. Furthermore, we show directly and constructively that the coinductive notion of u. c. is closed under composition. The extracted stream transformers are represented by finitely branching non-wellfounded trees which, if executed in a lazy programming language, give rise to memoized algorithms. These trees turn out to be closely related to the data structures studied in [21, 22], and the extracted program from the proof of closure under composition is a generalization of the tree composing program defined there.

In Sect. 2, we briefly review inductive and coinductive sets defined by monotone set operators. We give some simple examples, among them a characterization of the real numbers in the interval $\mathbb{I}$ by a coinductive predicate C_0. The method of program extraction from proofs involving induction and coinduction is discussed informally, but in some detail, in Sect. 3. The earlier examples are continued and a program transforming fast Cauchy representations into signed digit representations is extracted from a coinductive proof. In Sect. 4, the coinductive characterization C_0 of real numbers is generalized to nested coinductive/inductive predicates C_n characterizing uniformly continuous real functions of n arguments, and closure under composition is proven. In Sect. 5, we study wellfounded induction from the perspective of program extraction and introduce the notion of a *digital system* as a technical tool for showing that certain families of functions are contained in C_n. The positive effect of memoization is demonstrated by a case study on iterated logistic maps (which are special polynomials of degree 2). Furthermore, we prove that the predicates C_n capture precisely uniform continuity. In Sect. 6 we extract a program for integration from a proof that the definite integral on $\mathbb{I}$ of a function in C_1 can be approximated by rational numbers with any given precision.

The extracted programs are shown in the functional programming language Haskell. As Haskell's syntax is very close to the usual mathematical notation for data and functions we hope that also readers not familiar with Haskell will be able to understand the code. The Haskell code shown in this paper is self contained and can be obtained from the author on request.

2. Induction and coinduction

We briefly discuss inductive and coinductive definitions as least and greatest fixed points of monotone set operators and the corresponding induction and coinduction principles. The results in this section are standard and can be found in many logic and computer science texts. For example in [14] inductive definitions are proof-theoretically analysed, and in [13] least and greatest fixed points are studied in the framework of the modal mu-calculus.

An operator $\Phi\colon \mathcal{P}(U) \to \mathcal{P}(U)$ (where U is an arbitrary "universal" set and $\mathcal{P}(U)$ is the powerset of U) is *monotone* if for all $X, Y \subseteq U$

$$\text{if } X \subseteq Y, \text{ then } \Phi(X) \subseteq \Phi(Y)$$

A set $X \subseteq U$ is Φ-*closed* (or a pre-fixed point of Φ) if $\Phi(X) \subseteq X$. Since $\mathcal{P}(U)$ is a complete lattice, Φ has a least fixed point $\mu\Phi$ (Knaster-Tarski Theorem). For the sake of readability we will sometimes write $\mu X.\Phi(X)$ instead of $\mu\Phi$. $\mu\Phi$ can be defined as the least Φ-closed subset of U. Hence we have the *closure principle* for $\mu\Phi$, $\Phi(\mu\Phi) \subseteq \mu\Phi$ and the *induction principle* stating that for every $X \subseteq U$, if $\Phi(X) \subseteq X$, then $\mu\Phi \subseteq X$. It can easily be shown that $\mu\Phi$ is even a *fixed point* of Φ, i. e. $\Phi(\mu\Phi) = \mu\Phi$ (Lambek's Lemma). For monotone operators $\Phi, \Psi\colon \mathcal{P}(U) \to \mathcal{P}(U)$ we define

$$\Phi \subseteq \Psi \;\;:\Leftrightarrow\;\; \forall X \subseteq U\, \Phi(X) \subseteq \Psi(X)$$

It is easy to see that the operation μ *is monotone*, i. e. if $\Phi \subseteq \Psi$, then $\mu\Phi \subseteq \mu\Psi$. Using monotonicity of μ one can easily prove, by induction, a principle, called *strong induction*. It says that, if $\Phi(X \cap \mu\Phi) \subseteq X$, then $\mu\Phi \subseteq X$.

Dual to inductive definitions are *coinductive definitions*. A subset X of U is called Φ-*coclosed* (or a post-fixed point of Φ) if $X \subseteq \Phi(X)$. By duality, Φ has a largest fixed point $\nu\Phi$ which can be defined as the largest Φ-coclosed subset of U. Similarly, all other principles for induction have their coinductive counterparts. To summarise, we have the following principles:

Fixed point	$\Phi(\mu\Phi) = \mu\Phi$ and $\Phi(\nu\Phi) = \nu\Phi$.
Monotonicity	if $\Phi \subseteq \Psi$, then $\mu\Phi \subseteq \mu\Psi$ and $\nu\Phi \subseteq \nu\Psi$.
Induction	if $\Phi(X) \subseteq X$, then $\mu\Phi \subseteq X$.
Strong induction	if $\Phi(X \cap \mu\Phi) \subseteq X$, then $\mu\Phi \subseteq X$.
Coinduction	if $X \subseteq \Phi(X)$, then $X \subseteq \nu\Phi$.
Strong coinduction	if $X \subseteq \Phi(X \cup \nu\Phi)$, then $X \subseteq \nu\Phi$.

Example 2.1 (natural numbers). Define $\Phi : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{R})$ by

$$\Phi(X) := \{0\} \cup \{y + 1 \mid y \in X\} = \{x \mid x = 0 \vee \exists y \in X\, (x = y + 1)\}$$

Then $\mu\Phi = \mathbb{N} = \{0, 1, 2, \ldots\}$. We consider this as the *definition* of the natural numbers. The induction principle is logically equivalent to the usual zero-successor-induction on $\mathbb{N}$: if $X(0)$ (base) and $\forall x\, (X(x) \to X(x + 1))$ (step), then $\forall x \in \mathbb{N}\, X(x)$. Strong induction weakens the step by restricting x to the natural numbers: $\forall x \in \mathbb{N}\, (X(x) \to X(x + 1))$.

Example 2.2 (signed digits and the interval $[-1, 1]$). For every signed digit $d \in \mathrm{SD}$ we set $\mathbb{I}_d := [d/2 - 1/2, d/2 + 1/2] = \{x \in \mathbb{R} \mid |x - d/2| \leq 1/2\}$. Note that $\mathbb{I}$ is the union of the $\mathbb{I}_d$ and every sub interval of $\mathbb{I}$ of length $\leq 1/2$ is contained in some $\mathbb{I}_d$. We define an operator $\mathcal{J}_0 : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{R})$ by

$$\mathcal{J}_0(X) := \{x \mid \exists d \in \mathrm{SD}\, (x \in \mathbb{I}_d \wedge 2x - d \in X)\}$$

and set $\mathrm{C}_0 := \nu\mathcal{J}_0$. Since clearly $\mathbb{I} \subseteq \mathcal{J}_0(\mathbb{I})$, it follows, by coinduction, that $\mathbb{I} \subseteq \mathrm{C}_0$. On the other hand $\mathrm{C}_0 \subseteq \mathbb{I}$, by the fixed point property. Hence $\mathrm{C}_0 = \mathbb{I}$. The point of this definition is, that the proof of "$\mathbb{I} \subseteq \mathcal{J}_0(\mathbb{I})$" has an interesting computational content: $x \in \mathbb{I}$ must be given in such a way that it is possible to find $d \in \mathrm{SD}$ such that $x \in \mathbb{I}_d$. This means that $d/2$ is a *first approximation* of x. The computational content of the proof of "$\mathbb{I} \subseteq \mathrm{C}_0$", roughly speaking, iterates the process of finding approximations to x ad infinitum, i. e. it computes a *signed digit representation* of x as explained in the introduction, that is, a stream a of signed digits with $\sigma(a) = x$. This will be made precise in Lemma 3.2 (Sect. 4).

Example 2.3 (lists, streams and trees). Let the Scott-domain D be defined by the recursive domain equation $D = 1 + D \times D$ where $1 := \{\bot\}$ is a one point domain and "+" denotes the separated sum of domains (see [20] for information on domains). The elements of D are $\bot$ (the obligatory least element), $\mathrm{Nil} := \mathrm{Left}(\bot)$, and $\mathrm{Cons}(x, y) := \mathrm{Right}(x, y)$ where $x, y \in D$. Define $\Phi : \mathcal{P}(D) \to \mathcal{P}(D) \to \mathcal{P}(D)$ by

$$\Phi(X)(Y) := \{\mathrm{Nil}\} \cup \{\mathrm{Cons}(x, y) \mid x \in X, y \in Y\}$$

Clearly, Φ is monotone in both arguments. For a fixed set $X \subseteq D$, $\mathrm{List}(X) := \mu(\Phi(X))$ $(= \mu Y.\Phi(X)(Y))$ can be viewed as the set of *finite* lists of elements in X, and $\mathrm{Stream}(X) :=$ $\nu(\Phi(X))$ $(= \nu Y.\Phi(X)(Y))$ as the set of *finite or infinite* lists or *streams* of elements in X. Since μ is monotone the operator $\mathrm{List} : \mathcal{P}(D) \to \mathcal{P}(D)$ is again monotone. Hence we can define $\mathrm{Tree} := \nu\mathrm{List} \subseteq D$ which is the set of finitely branching wellfounded or non-wellfounded trees. On the other hand, $\mathrm{Tree}' := \mu\mathrm{Stream}$ consist of all finitely or infinitely branching wellfounded trees. The point of this example is that the definition of Tree is similar to the characterization of uniformly continuous functions from $\mathbb{I}^n$ to $\mathbb{I}$ in Sect. 4, the similarity being the fact that it is a coinductive definition with an inductive definition in its body. The set C_0 of the previous example corresponds to the case $n = 0$ where the inner inductive definition is trivial.

Formalization We now sketch the formal system for reasoning about inductive and coinductive definitions (a full account is given in [7, 10]). Since we only consider (co)inductive definitions of subsets of a given "universal set" we can work in a many-sorted first-order predicate logic with free predicate variables extended by the possibility to form for a predicate $\mathcal{P}$ which is strictly positive (s.p.) in a predicate variable X the predicates $\mu X.\mathcal{P}$ and $\nu X.\mathcal{P}$ denoting the least and greatest fixed points of the monotone set operator defined by $\mathcal{P}$. For example, $\mathcal{P}$ could be given as a comprehension term $\{\vec{x} \mid A(\vec{x}, X)\}$ where $A(\vec{x}, X)$ is a formula which is s.p. in X. The formula $A(\vec{x}, X)$ may have further free object and predicate variables. "Nested" inductions/coinductions such as $\nu X.\mu Y.\{\vec{x} \mid A(\vec{x}, X, Y)\}$, where $A(\vec{x}, X, Y)$ is strictly positive in X and Y, are allowed. Hence the second example above can be formalized. As a proof calculus we use intuitionistic natural deduction with axioms expressing (co)closure and (co)induction for (co)inductively defined predicates. Further axioms describing the mathematical structures under consideration can be freely added as long as (we know that) they are true and do not contain disjunctions. The latter restriction ensures that these "ad-hoc axioms" have no computational content, as will be explained in Sect. 3. Note that, for example, the formula $\forall x \in \mathbb{N}\, \exists y \in \mathbb{N}\, (y^2 \leq x < (y + 1)^2)$ *does* have computational content since the definition of the predicate $\mathbb{N}$, given in the first example, contains a disjunction. Hence, although true, this formula must not be used as an axiom, but needs to be proven.

In the first example object variables are of sort $\mathbb{R}$ while in the second example they are of sort D (hence $\mathbb{R}$ and D are the "universal sets"). The second example shows how data structures which normally would be defined as initial algebras or final coalgebras of endofunctors on the category of sets can be introduced in our system. The domain-theoretic modelling has the further advantage that partial objects (e.g. lists or trees with possibly undefined nodes and leaves) can be described and reasoned about as well, without extra effort. We believe that, by restricting ourselves to categories which are just powersets, partially ordered by inclusion, the constructions become easier to understand for non-category-theorists, and the formal system sketched above is simpler than one describing initial algebras and final coalgebras of functors in general.

3. Program extraction from proofs

In this section we briefly explain how we extract programs from proofs. Rather than giving technical definitions we only sketch the formal framework and explain the extraction method by means of simple examples, which hopefully provide a good intuition also for non-experts. More details and full correctness proofs can be found in [7] and [10].

The method of program extraction we are using is based on an extension and variation of Kreisel's *modified realizability* [26]. The *extension* concerns the addition of inductive and coinductive predicates. Realizability for such predicates has been studied previously, in the slightly different context of **q**-realizability by Tatsuta [36]. The *variation* concerns the fact that we are treating the first-order part of the language (i. e. quantification over individuals) in a 'uniform' way, that is, realizers do not depend on the individuals quantified over. This is similar to the common uniform treatment of second-order variables [37]. The argument is that an arbitrary subset of a set is such an abstract (and even vague) entity so that one should not expect an algorithm to depend on it. With a similar argument one may say that individuals of an abstract mathematical structure ($\mathbb{R}$, model of set-theory, etc.) are unsuitable as inputs for programs. Hence, a realizer of a formula $\forall x\, A(x)$ is an object a such that a realizes $A(x)$ for *all* x where a does not depend on x. A realizer of a formula $\exists x\, A(x)$ is an object a such that a realizes $A(x)$ for *some* x. Note that the witness x is not part of the realizer a. But which data should a program then depend on and which should it produce? The answer is: data defined by the 'propositional skeletons' of formulas and 'canonical' proofs.

Example (parity) Let us extract a program from a proof of

$$\forall x\, (\mathbb{N}(x) \Rightarrow \exists y\, (x = 2y \lor x = 2y + 1)) \tag{3.1}$$

where the variable x ranges over real numbers and the predicate $\mathbb{N}$ is defined as in the example in Sect. 2, i. e.

$$\mathbb{N} := \mu X.\{x \mid x = 0 \lor \exists y\, (X(y) \land x = y + 1)\} \tag{3.2}$$

The type corresponding to (3.2) is obtained by the following *type extraction*:

- replace every atomic formula of the form $X(t)$ by a type variable α associated with the predicate variable X,
- replace other atomic formulas by the unit or 'void' type $\mathbf{1}$,
- delete all quantifiers and object terms (i. o. w. remove all first-order parts),
- replace $\lor$ by $+$ (disjoint sum) and $\land$ by $\times$ (cartesian product),
- carry out obvious simplifications (e.g. replace $\alpha \times \mathbf{1}$ by α).

Hence we arrive at the type definition

$$\mathrm{Nat} := \mu\alpha.\mathbf{1} + \alpha$$

In Haskell we can define this type as

```
data Nat = Zero | Succ Nat      -- data
   deriving Show
```

The line "`deriving Show`" creates a default printing method for values of type `Nat`. The comment "`-- data`" indicates that we intend to use the recursive data type `Nat` as an inductive data type (or initial algebra). This means that the "'total", or "legal" elements are inductively generated from `Zero` and `Succ`. The natural (domain-theoretic) semantics of `Nat` also contains, for example, an "infinite" element defined recursively by `infty = Succ infty` which is not total in the inductive interpretation of `Nat`. In a coinductive interpretation (usually indicated by the comment `-- codata`) `infty` would count as total.[1]

By applying type extraction to (3.1) we see that a program extracted from a proof of this formula will have type $\mathrm{Nat} \to \mathbf{1} + \mathbf{1}$. By identifying the two-element type $\mathbf{1} + \mathbf{1}$ with the Booleans we get the Haskell signature

```
parity :: Nat -> Bool
```

The definition of `parity` can be extracted from the obvious inductive proof of (3.1): For the base, $x = 0$, we take $y = 0$ to get $x = 2y$. In the step, $x + 1$, we have, by i. h. some y with $x = 2y \vee x = 2y + 1$. In the first case $x + 1 = 2y + 1$, in the second case $x + 1 = 2(y + 1)$. The Haskell program extracted from this proof is

```
parity Zero     = True
parity (Succ x) = case parity x of {True -> False ; False -> True}
```

If we wish to compute not only the parity, but as well the rounded down half of x (i. e. quotient and remainder), we just need to relativize the quantifier $\exists y$ in (3.1) to $\mathbb{N}$ (i. e. $\forall x\,(\mathbb{N}(x) \Rightarrow \exists y\,(\mathbb{N}(y) \wedge (x = 2y \vee x = 2y + 1)))$ and use in the proof the fact that $\mathbb{N}$ is closed under the successor operation. The extracted program is then

```
parity1 :: Nat -> (Nat,Bool)
parity1 Zero     = (Zero,True)
parity1 (Succ x) = case parity1 x of
                    {(y,True)  -> (y,False) ;
                     (y,False) -> (Succ y,True)}
```

In order to try these programs out it is convenient to have a function that transforms built-in integers into elements of `Nat`.

```
iN :: Integer -> Nat  -- defined for non-negative integers only
iN 0     = Zero
iN (n+1) = Succ (iN n)
```

Now try `parity (iN 7)` and `parity1 (iN 7)`.

The examples above show that we can get meaningful computational content despite ignoring the first-order part of a proof. Moreover, we can fine-tune the amount of computational information we extract from a proof by simple modifications of formulas and proofs. Note also that we used arithmetic operations on the reals and their arithmetic laws

[1] That Haskell does not distinguish between the inductive and the coinductive interpretation is justified by the limit-colimit-coincidence in the domain-theoretic semantics [1].

without implementing or proving them. Since these laws can be written as equations (or conditional equations) their associated type is void. This ensures that it is only their truth that matters, allowing us to treat them as ad-hoc axioms without bothering to derive them from basic axioms. In general, a formula containing neither disjunctions nor free predicate variables has always a void type and can therefore be taken as an axiom as long as it is true.

The reader might be puzzled by the fact that quantifiers are ignored in the program extraction process. Quantifiers are, of course, *not* ignored in the *specification* of the extracted program, i. e. in the definition of realizability. For example, the statement that the program $p :=$`parity` realizes (3.1) is expressed by

$$\forall n, x\,(n\,\mathbf{r}\,\mathbb{N}(x) \Rightarrow \exists y\,(p(n) = \texttt{True} \land x = 2y \lor p(n) = \texttt{False} \land x = 2y + 1))$$

where n ranges over Nat (i. e. `Zero`, `Succ Zero`, `Succ(Succ Zero)`, ...) and $n\,\mathbf{r}\,\mathbb{N}(x)$ means that n realizes $\mathbb{N}(x)$ which in this case amounts to x being the value of n in $\mathbb{R}$. The *Soundness Theorem* for realizability states that the program extracted from a proof realizes the proven formula (cf. [10]; see also e.g. [37], [36] for proofs of soundness for related notions of realizability).

Remark. Although the example above seems to suggest that realizers are typed, it is in fact more convenient to work with per se untyped realizers taken from a domain D which is defined by the recursive domain equation

$$D = 1 + D + D + D \times D + [D \to D]$$

($[D \to D]$ denotes the domain of continuous endofunctions on D). It is well-known that such domain equations of "mixed variance" have effective solutions up to isomorphism (see e.g. [20]). Type expressions $\mathbf{1}$, α, $\rho + \sigma$, $\rho \times \sigma$, $\rho \to \sigma$, $\mu\alpha.\rho$, $\nu\alpha.\rho$ with suitable positivity conditions for fixed point types can naturally be interpreted as subsets of D.[2] Realizers extracted from proofs are terms of an untyped λ-calculus with constructors and recursion which denote elements of D. It can be shown that the value of a program extracted from a proof of a formula A lies in the denotation of the type extracted from A [10]. One can also show that the denotational and operational semantics "match" (computational adequacy). This implies that extracted programs are correct, both in a denotational and operational sense [7]. Note that in general a realizing term denotes an element of D, but not an element of the mathematical structure the proof is about. It is just by coincidence that in the example above the closed terms of type Nat denote at the same time elements of D and real numbers, and that both denotations are in a one-to-one correspondence. In the case of the predicate C_0 defined below (and even more so for the predicates C_n defined in Sect. 4) there is no such tight correspondence between objects satisfying a predicate and realizers of that fact.

Example 3.1 (from Cauchy sequences to signed digit streams). In the second example of Sect. 2 we defined the set C_0 coinductively by

$$\mathrm{C}_0 = \nu X.\{x \mid \exists d\,(\mathrm{SD}(d) \land \mathbb{I}_d(x) \land X(2x - d))\} \tag{3.3}$$

[2] In fact, general recursive types $\mathrm{rec}\,\alpha\,.\,\rho$ without positivity restriction have a natural semantics in D as (ranges of) finitary projections [3].

Since $\mathrm{SD}(d)$ is shorthand for $d = -1 \lor d = 0 \lor d = 1$, and $\mathbb{I}_d(x)$ is shorthand for $|x - d/2| \leq 1/2$, the corresponding type is

$$\mathrm{C0} = \nu\alpha.(\mathbf{1} + \mathbf{1} + \mathbf{1}) \times \alpha \tag{3.4}$$

Identifying notationally the type $\mathbf{1} + \mathbf{1} + \mathbf{1}$ with SD

```
data SD  = N | Z | P  -- Negative, Zero, Positive
   deriving Show
```

we obtain that C0 is the type of infinite streams of signed digits, i. e. the largest fixed point of the type operator

```
type J0 alpha = (SD,alpha)
```

This corresponds to the set operator $\mathcal{J}_0$ which C_0 is the largest fixed point of. Therefore we define (choosing `ConsC0` as constructor name)

```
data C0 = ConsC0 (J0 C0)    -- codata
```

i. e. `C0 = ConsC0 (SD,C0)`.

We wish to extract a program that computes a signed digit representation of $x \in \mathbb{I}$ from a fast rational Cauchy sequence converging to x and vice versa. Set

$$\begin{aligned} \mathbb{Q}(x) &:= \exists n, m, k\, (\mathbb{N}(n) \land \mathbb{N}(m) \land \mathbb{N}(k) \land x = (n-m)/k) \\ A(x) &:= \forall n\, (\mathbb{N}(n) \Rightarrow \exists q\, (\mathbb{Q}(q) \land |x - q| \leq 2^{-n})) \end{aligned}$$

Constructively, $A(x)$ means that there is a fast Cauchy sequence of rational numbers converging to x. Technically, this is expressed by the fact that the realizers of $A(x)$ are precisely such sequences. On the other hand, realizers of $\mathrm{C}_0(x)$ are exactly the infinite streams of signed digits a such that $\sigma(a) = x$ In general, realizability for inductive resp. coinductive predicates is defined in a straightforward way, again as an inductive resp. coinductive definition (see [7, 10] for details).

Lemma 3.2.

$$\forall x\, (\mathbb{I}(x) \land A(x) \Leftrightarrow \mathrm{C}_0(x)) \tag{3.5}$$

Proof. To prove the implication from left to right we show $\mathbb{I} \cap A \subseteq \mathrm{C}_0$ by coinduction, i. e. we show $\mathbb{I} \cap A \subseteq \mathcal{J}_0(\mathbb{I} \cap A)$. Assume $\mathbb{I}(x)$ and $A(x)$. We have to show (constructively!) $\mathcal{J}_0(\mathbb{I} \cap A)(x)$, i. e. we need to find $d \in \mathrm{SD}$ such that $x \in \mathbb{I}_d$ and $2x - d \in \mathbb{I} \cap A$. Since, clearly the assumption $A(x)$ implies $A(2x - d)$ for any $d \in \mathrm{SD}$, and furthermore $x \in \mathbb{I}_d$ holds iff $2x - d \in \mathbb{I}$, we only need to find some signed digit d such that $x \in \mathbb{I}_d$. The assumption $A(x)$, used with $n = 2$, yields a rational number q with $|x - q| \leq 1/4$. It is easy to find (constructively!) a signed digit d such that $[q - 1/4, q + 1/4] \cap \mathbb{I} \subseteq \mathbb{I}_d$. For that d we have $x \in \mathbb{I}_d$.

For the converse implication we show $\forall n\, (\mathbb{N}(n) \Rightarrow \forall x\, (\mathrm{C}_0(x) \Rightarrow \exists q\, (\mathbb{Q}(q) \land |x - q| \leq 2^{-n}))$ by induction on $\mathbb{N}(n)$ using the coclosure axiom for C_0. We leave the details as an exercise for the reader. □

The type corresponding to the predicate $\mathbb{Q}$ is Nat × Nat × Nat, which we however implement by Haskell's built-in rationals, since it is only the arithmetic operations on rational numbers that matter, whatever the representation. (It is possible - and instructive as an exercise - to extract implementations of the arithmetic operations on rational numbers w.r.t. the representation Nat × Nat × Nat from proofs that $\mathbb{Q}$ is closed under these operations. In order to obtain reasonably efficient programs one has to modify the definition of $\mathbb{Q}$ by requiring $n-m$ and k to be relatively prime.) The type of the predicate A is Nat → Rational. The program extracted from the first part of the proof of Lemma 3.2 is

```
cauchy2sd :: (Nat -> Rational) -> C0
cauchy2sd = coitC0 step
```

where `step` is the program extracted from the proof of $\mathbb{I} \cap A \subseteq \mathcal{J}_0(\mathbb{I} \cap A)$:

```
step :: (Nat -> Rational) -> J0(Nat -> Rational)
step f = (d,f')  where
  q = f (Succ (Succ Zero))
  d = if q > 1/4 then P else if abs q <= 1/4 then Z else N
  f' n = 2 * f (Succ n) - fromSD d

fromSD :: SD -> Rational
fromSD d = case d of {N -> -1 ; Z -> 0 ; P -> 1}
```

The program `coitC0` is a polymorphic "coiterator" realizing the coinduction scheme $X \subseteq \mathcal{J}_0(X) \Rightarrow X \subseteq \nu\mathcal{J}_0$:

```
coitC0 :: (alpha -> J0 alpha) -> alpha -> C0
coitC0 s x = ConsC0 (mapJ0 (coitC0 s) (s x))

mapJ0 :: (alpha -> beta) -> J0 alpha -> J0 beta
mapJ0 f (d,x) = (d,f x)
```

An equivalent definition of `coitC0` would be

```
coitC0' s x = ConsC0 (d,coitC0' s y) where (d,y) = s x
```

The program extracted from the second part of the proof of Lemma 3.2 is

```
sd2cauchy :: C0 -> (Nat -> Rational)
sd2cauchy c n = aux n c  where
  aux Zero c = 0
  aux (Succ n) (ConsC0 (d,c)) = (fromSD d + aux n c)/2
```

In order to try out the programs `cauchy2sd` and `sd2cauchy` it is convenient to have translations between the types `C0` and Haskell's type of infinite streams of signed digits (below, "`:`" is the cons operation for lists).

```
c0s :: C0 -> [SD]
c0s (ConsC0 (d,c)) = d : c0s c

sc0 :: [SD] -> C0
sc0 (d:ds) = ConsC0 (d,sc0 ds)
```

Now evaluate `let {f x = 2/3} in take 10 (c0s(cauchy2sd f))` and
`let {ds = P:Z:ds} in [sd2cauchy (sc0 ds) (iN n) | n <- [0..9]]`
(`ds` is the infinite list `[P,Z,P,Z,...]` and `[e(n) | n <- [0..9]]` is a list comprehension expression denoting `[e(0),...,e(n)]`).

We hope that the examples above give enough hints for understanding program extraction from coinductive proofs. Here is a sketch of how it works in general. Suppose $\nu\Phi$ is a coinductive predicate defined by a strictly positive set operator Φ ($\mathcal{J}_0$ in our example), e.g. $\Phi(X) = \{\vec{x} \mid A(X,\vec{x})\}$ where A is s.p. in X. From Φ one extracts a s.p. type operator

```
data Phi alpha = PhiDef(alpha)  -- to be replaced by a suitable
                                -- extracted type definition
```

(`J0` in our example). Due to the strict positivity of `Phi` one can define, by structural recursion on the definition of `Phi alpha`, a polymorphic map operation

```
mapPhi :: (alpha -> beta) -> Phi alpha -> Phi beta
mapPhi = undefined -- to be replaced by an extracted program
```

and from that, recursively, the coiterator

```
coitFix :: (alpha -> Phi alpha) -> alpha -> Fix
coitFix s x = ConsFix (mapPhi (coitFix s) (s x))
```

where `Fix` is the largest fixed point of `Phi`:

```
data Fix = ConsFix (Phi Fix)  -- codata
```

The program extracted from a coinductive proof of $X \subseteq \nu\Phi$ is `coitFix step` where `step :: alpha -> Phi alpha` is the program extracted from the proof of $X \subseteq \Phi(X)$ (`alpha` is the type corresponding to the predicate X). For inductive proofs the construction is similar: One defines recursively an "iterator"

```
itFix :: (Phi alpha -> alpha) -> Fix -> alpha
itFix s (ConsFix z) = s (mapPhi (itFix s) z)
```

where the type `Fix` is now viewed as the *least* fixed point of `Phi`. The program extracted from an inductive proof of $\mu\Phi \subseteq X$ is now `itFix step` where `step :: Phi alpha -> alpha` is extracted from the proof of $\Phi(X) \subseteq X$. It is a useful exercise to re-program the data type `Nat` and the iteratively defined functions `parity`, `parity1` and `sd2cauchy` following strictly this general scheme. The above sketched computational interpretations of induction and coinduction and more general recursive schemes can be derived from category-theoretic considerations using the initial algebra/final coalgebra interpretation of least and greatest fixed points (see for example [28, 23, 2, 15]).

4. Coinductive definition of uniform continuity

For every n we define a set $\mathrm{C}_n \subseteq \mathbb{R}^{\mathbb{I}^n}$ for which we will in Sect. 5 show that it coincides with the set of uniformly continuous functions from $\mathbb{I}^n$ to $\mathbb{I}$.

In the following we let n, m, k, l, i range over $\mathbb{N}$, p, q over $\mathbb{Q}$, x, y, z over $\mathbb{R}$, and d, e over SD. Hence, for example, $\exists d\, A(d)$ is shorthand for $\exists d\, (\mathrm{SD}(d) \wedge A(d))$ and $\bigwedge_d A(d)$ abbreviates $A(-1) \wedge A(0) \wedge A(1)$. We define average functions and their inverses

$$\begin{aligned}\mathrm{av}_d\colon \mathbb{R} \to \mathbb{R}, \qquad & \mathrm{av}_d(x) := \frac{x+d}{2}\\ \mathrm{va}_d\colon \mathbb{R} \to \mathbb{R}, \qquad & \mathrm{va}_d(x) := 2x - d\end{aligned}$$

Note that $\mathrm{av}_d[\mathbb{I}] = \mathbb{I}_d$ and hence $f[\mathbb{I}] \subseteq \mathbb{I}_d$ iff $(\mathrm{va}_d \circ f)[\mathbb{I}] \subseteq \mathbb{I}$. We also need extensions of the average functions to n-tuples

$$\mathrm{av}_{i,d}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) := (x_1, \ldots, x_{i-1}, \mathrm{av}_d(x_i), x_{i+1}, \ldots, x_n)$$

We define an operator $\mathcal{K}_n\colon \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n})$ by

$$\mathcal{K}_n(X)(Y) := \{f \mid \exists d\, (f[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge X(\mathrm{va}_d \circ f)) \vee \exists i \bigwedge_d Y(f \circ \mathrm{av}_{i,d})\}$$

Since $\mathcal{K}_n$ is strictly positive in both arguments, we can define an operator $\mathcal{J}_n \colon \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n})$ by

$$\mathcal{J}_n(X) := \mu(\mathcal{K}_n(X)) = \mu Y.\mathcal{K}_n(X)(Y)$$

Hence, $\mathcal{J}_n(X)$ is the set inductively defined by the following two rules:

$$\exists d\,(f[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge X(\mathrm{va}_d \circ f)) \Rightarrow \mathcal{J}_n(X)(f) \tag{4.1}$$

$$\exists i \bigwedge_d \mathcal{J}_n(X)(f \circ \mathrm{av}_{i,d}) \Rightarrow \mathcal{J}_n(X)(f) \tag{4.2}$$

Since, as mentioned in Sect. 2, the operation μ is monotone, $\mathcal{J}_n$ is monotone as well. Therefore, we can define C_n as the largest fixed point of $\mathcal{J}_n$,

$$\mathrm{C}_n = \nu\mathcal{J}_n = \nu X.\mu Y.\mathcal{K}_n(X)(Y) \tag{4.3}$$

Note that for $n = 0$ the second argument Y of $\mathcal{K}_n$ becomes a dummy variable, and therefore $\mathcal{J}_0$ and C_0 are the same as in the corresponding example in Sect. 2. Note also that if $f \in \mathrm{C}_n$, then $f[\mathbb{I}^n] \subseteq \mathbb{I}_d \subseteq \mathbb{I}$ for some $d \in \mathrm{SD}$ since $\mathrm{C}_n = \mu Y.\mathcal{K}_n(\mathrm{C}_n)(Y) = \mathcal{K}_n(\mathrm{C}_n)(\mathrm{C}_n)$.

The type corresponding to the formula $\mathcal{K}_n(X)(Y)$ is $\varphi_n(\alpha)(\beta) := \mathrm{SD} \times \alpha + \mathbb{N}_n \times \beta^3$ where $\mathbb{N}_n := \{1, ..., n\}$. Therefore, the type of $\mathcal{J}_n(X)$ is $\mu\beta.\mathrm{SD} \times \alpha + \mathbb{N}_n \times \beta^3$ which is the type of finite ternary trees with indices $i \in \mathbb{N}_n$ attached to the inner nodes and pairs $(d, x) \in \mathrm{SD} \times \alpha$ attached to the leaves. Consequently, the type of C_n is

$$\nu\alpha.\mu\beta.\mathrm{SD} \times \alpha + \mathbb{N}_n \times \beta^3 \tag{4.4}$$

This is the type of non-wellfounded trees obtained by infinitely often stacking the finite trees on top of each other, i. e. replacing in a finite tree each x in a leaf by another finite tree and repeating the process in the substituted trees ad infinitum. Alternatively, the elements of (4.4) can be described as non-wellfounded trees without leaves such that

- each node is either a
  writing node labelled with a signed digit and with one subtree, or a
  reading node labelled with an index $i \in \mathbb{N}_n$ and with three subtrees;
- each path has infinitely many writing nodes.

The interpretation of such a tree as a stream transformer is easy. Given n signed digit streams $a_1, \ldots, a_n$ as inputs, run through the tree and output a signed digit stream as follows:

1. At a writing node (d, t) output d and continue with the subtree t.
2. At a reading node $(i, (t_d)_{d \in \mathrm{SD}})$ continue with t_d, where d is the head of a_i, and replace a_i by its tail.

Fig. 1 shows an initial segment of a tree representing the function

$$f \colon \mathbb{I} \to \mathbb{I}, \quad f(x) = \frac{2}{3}(1 - x^2) - 1$$

which is an instance of the family of logistic maps discussed in Sect. 5. In order to "run" this tree with an input stream of signed digits, we follow the path determined by the input digits. `N`, `Z` or `P` in the input stream means: go at a branching point left, middle or right. The digits met on this path form the output stream. For example, the input stream `Z:Z:Z:Z:...` (representing the number 0) leads us along the spine of the tree and results in the output stream `N:Z;P:Z:P:Z:...` (representing $\frac{-1}{2} + \frac{1}{8} + \frac{1}{32} + \ldots = \frac{-1}{2} + \frac{1}{6} = -\frac{1}{3} = f(0)$) while

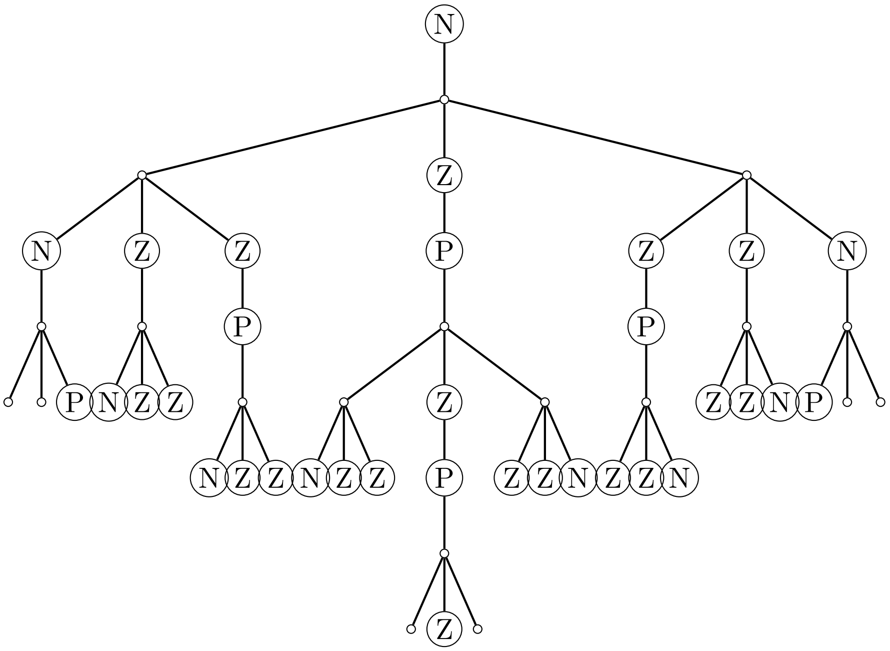

Figure 1: An initial segment of the tree of $f(x) = \frac{2}{3}(1 - x^2) - 1$.

the input stream `P:Z:Z:Z:...` (representing the number $\frac{1}{2}$) results in the output stream `N:Z;Z:Z:...` (representing $-\frac{1}{2} = f(\frac{1}{2})$). [3]

The above informally described interpretation of the elements of C_n as stream transformers is the extracted program of a special case of Proposition 4.2 below which shows that the predicates C_n are closed under composition. The following lemma is needed in its proof.

Lemma 4.1. *If* $\mathrm{C}_n(f)$*, then* $\mathrm{C}_n(f \circ \mathrm{av}_{i,d})$*.*

Proof. We fix $i \in \{1, \dots, n\}$ and $d \in \mathrm{SD}$ and set

$$D := \{f \circ \mathrm{av}_{i,d} \mid \mathrm{C}_n(f)\}$$

We show $D \subseteq \mathrm{C}_n$ by strong coinduction, i. e. we show $D \subseteq \mathcal{J}_n(D \cup \mathrm{C}_n)$, i. e. $\mathrm{C}_n \subseteq E$ where

$$E := \{f \mid \mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})\}$$

Since $\mathrm{C}_n = \mathcal{J}_n(\mathrm{C}_n)$ it suffices to show $\mathcal{J}_n(\mathrm{C}_n) \subseteq E$. We prove this by strong induction on $\mathcal{J}_n(\mathrm{C}_n)$, i. e. we show $\mathcal{K}_n(\mathrm{C}_n)(E \cap \mathcal{J}_n(\mathrm{C}_n)) \subseteq E$. Induction base: Assume $f[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$ and $\mathrm{C}_n(\mathrm{va}_{d'} \circ f)$. We need to show $E(f)$, i. e. $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$. By (4.1) it suffices to show $(f \circ \mathrm{av}_{i,d})[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$ and $(D \cup \mathrm{C}_n)(\mathrm{va}_{d'} \circ f \circ \mathrm{av}_{i,d})$. We have $(f \circ \mathrm{av}_{i,d})[\mathbb{I}^n] = f[\mathrm{av}_{i,d}[\mathbb{I}^n]] \subseteq f[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$. Furthermore, $D(\mathrm{va}_{d'} \circ f \circ \mathrm{av}_{i,d})$ holds by the assumption $\mathrm{C}_n(\mathrm{va}_{d'} \circ f)$ and the definition of D. Induction step: Assume, as strong induction hypothesis, $\bigwedge_{d'}(E \cap \mathcal{J}_n(\mathrm{C}_n))(f \circ \mathrm{av}_{i',d'})$. We have to show $E(f)$, i. e. $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$. If $i' = i$, then the strong induction hypothesis implies $\mathcal{J}_n(\mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$ which, by the monotonicity of $\mathcal{J}_n$, in turn implies $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$. If $i' \neq i$, then $\bigwedge_{d'} \mathrm{av}_{i',d'} \circ \mathrm{av}_{i,d} = \mathrm{av}_{i,d} \circ \mathrm{av}_{i',d'}$ and therefore, since

[3] The LaTeXcode for the display of the tree was generated automatically from a term denoting this tree which in turn was extracted from a formal proof that the function f lies in C_1.

the strong induction hypothesis implies $\bigwedge_{d'} E(f \circ \mathrm{av}_{i',d'})$, we have $\bigwedge_{d'} \mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d} \circ \mathrm{av}_{i',d'})$. By (4.2) this implies $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$. □

Proposition 4.2. *Consider $f\colon \mathbb{I}^n \to \mathbb{R}$ and $g_i\colon \mathbb{I}^m \to \mathbb{R}$, for $i = 1, \ldots, n$. If $\mathrm{C}_n(f)$ and $\mathrm{C}_m(g_1), \ldots, \mathrm{C}_m(g_n)$, then $\mathrm{C}_m(f \circ (g_1, \ldots, g_n))$.*

Proof. We prove the proposition by coinduction, i. e. we set

$$D := \{f \circ (g_1, \ldots, g_n) \mid \mathrm{C}_n(f),\ \mathrm{C}_m(g_1),\ \ldots,\ \mathrm{C}_m(g_n)\}$$

and show that $D \subseteq \mathcal{J}_m(D)$, i. e. $\mathrm{C}_n \subseteq E$ where

$$E := \{f \in \mathbb{R}^{\mathbb{I}^n} \mid \forall \vec{g}\,(\mathrm{C}_m(\vec{g}) \Rightarrow \mathcal{J}_m(D)(f \circ \vec{g}))\}$$

and $\mathrm{C}_m(\vec{g}) := \mathrm{C}_m(g_1) \wedge \ldots \wedge \mathrm{C}_m(g_n)$. Since $\mathrm{C}_n = \mathcal{J}_n(\mathrm{C}_n)$ it suffices to show $\mathcal{J}_n(\mathrm{C}_n) \subseteq E$. We do an induction on $\mathcal{J}_n(\mathrm{C}_n)$, i. e. we show $\mathcal{K}_n(\mathrm{C}_n)(E) \subseteq E$. Induction base: Assume $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$, $\mathrm{C}_n(\mathrm{va}_d \circ f)$ and $\mathrm{C}_m(\vec{g})$. We have to show $\mathcal{J}_m(D)(f \circ \vec{g}))$. By (4.1) it suffices to show $(f \circ \vec{g})[\mathbb{I}^m] \subseteq \mathbb{I}_d$ and $D(\mathrm{va}_d \circ f \circ \vec{g})$. The first statement holds since $\vec{g}[\mathbb{I}^m] \subseteq \mathbb{I}$, the second holds by the definition of D and the assumption. Induction step: Assume, as induction hypothesis, $\bigwedge_d E(f \circ \mathrm{av}_{i,d})$. We have to show $E(f)$, i. e. $\mathrm{C}_m \subseteq F$ where

$$F := \{g \in \mathbb{R}^{\mathbb{I}^m} \mid \forall \vec{g}\,(g = g_i \wedge \mathrm{C}_m(\vec{g}) \Rightarrow \mathcal{J}_m(D)(f \circ \vec{g}))\}$$

Since $\mathrm{C}_m \subseteq \mathcal{J}_m(\mathrm{C}_m)$ it suffices to show $\mathcal{J}_m(\mathrm{C}_m) \subseteq F$ which we do by a side induction on $\mathcal{J}_m$, i. e. we show $\mathcal{K}_m(\mathrm{C}_m)(F) \subseteq F$. Side induction base: Assume $g[\mathbb{I}^m] \subseteq \mathbb{I}_d$ and $\mathrm{C}_m(\mathrm{va}_d \circ g)$ and $\mathrm{C}_m(\vec{g})$ where $g = g_i$. We have to show $\mathcal{J}_m(D)(f \circ \vec{g})$. Let $\vec{g}'$ be obtained from $\vec{g}$ by replacing g_i with $\mathrm{va}_d \circ g$. Since $\mathrm{C}_m(\vec{g}')$, we have $\mathcal{J}_m(D)(f \circ \mathrm{av}_{i,d} \circ \vec{g}')$, by the main induction hypothesis. But $\mathrm{av}_{i,d} \circ \vec{g}' = \vec{g}$. Side induction step: Assume $\bigwedge_d F(g \circ \mathrm{av}_{j,d})$ (side induction hypothesis). We have to show $F(g)$. Assume $\mathrm{C}_m(\vec{g})$ where $g = g_i$. We have to show $\mathcal{J}_m(D)(f \circ \vec{g})$. By (4.2) it suffices to show $\mathcal{J}_m(D)(f \circ \vec{g} \circ \mathrm{av}_{j,d})$ for all d. Since the i-th element of $\vec{g} \circ \mathrm{av}_{j,d}$ is $g \circ \mathrm{av}_{j,d}$ and, by Lemma 4.1, $\mathrm{C}_m(\vec{g} \circ \mathrm{av}_{j,d})$, we can apply the side induction hypothesis. □

The program extracted from Prop. 4.2 composes trees. The cases $m = 0$ and $n = 1$ are of particular interest. If $m = 0$, then the program interprets a tree in C_n as an n-ary stream transformer. In the proof the functions $\vec{g}$ are then just real numbers, and composition, $f \circ \vec{g}$, becomes function application, $f(\vec{g})$. Furthermore, the side induction step disappears. If $n = 1$, then the vectors $\vec{g}$ consist of only one function g and F simplifies to $\{g \in \mathbb{R}^{\mathbb{I}^m} \mid \mathcal{J}_m(D)(f \circ g)\}$. Furthermore, the side induction step does not need Lemma 4.1 anymore and becomes almost trivial. We show the programs for the cases $n = 1, m = 0$ and $n = m = 1$. We use the following auxiliary programs extracted from a proof of the formula $(X(-1) \wedge X(0) \wedge X(1)) \Leftrightarrow \forall d\, X(d)$.

```
type Triple alpha = (alpha,alpha,alpha)

appTriple :: Triple alpha -> SD -> alpha
appTriple (xN,xZ,xP) d = case d of {N -> xN ; Z -> xZ ; P -> xP}

abstTriple :: (SD -> alpha) -> Triple alpha
abstTriple f = (f N, f Z, f P)
```

The data types associated with the operators $\mathcal{K}_1$, $\mathcal{J}_1$ and the predicate C_1 as well as their associated map functions and (co)iterators are

```
data K1 alpha beta = W1 SD alpha | R1 (Triple beta)

mapK1 :: (alpha -> alpha') -> (beta -> beta') ->
         K1 alpha beta -> K1 alpha' beta'
mapK1 f g (W1 d a) = W1 d (f a)
mapK1 f g (R1 (bN,bZ,bP)) = R1 (g bN,g bZ,g bP)

data J1 alpha = ConsJ1 (K1 alpha (J1 alpha))   -- data

itJ1 :: (K1 alpha beta -> beta) -> J1 alpha -> beta
itJ1 s (ConsJ1 z) = s (mapK1 id (itJ1 s) z)

mapJ1 :: (alpha -> alpha') -> J1 alpha -> J1 alpha'
mapJ1 f (ConsJ1 x) = ConsJ1 (mapK1 f (mapJ1 f) x)

data C1 = ConsC1 (J1 C1)      -- codata

coitC1 :: (alpha -> J1 alpha) -> alpha -> C1
coitC1 s x = ConsC1 (mapJ1 (coitC1 s) (s x))
```

Now, the extracted programs of Proposition 4.2. Case $n = 1, m = 0$:

```
appC :: C1 -> C0 -> C0
appC c ds = coitC0 costep (c,ds)  where

   costep :: (C1,C0) -> J0 (C1,C0)
   costep (ConsC1 x,ds) = aux x ds

   aux :: J1 C1 -> C0 -> J0 (C1,C0)
   aux = itJ1 step

   step :: K1 C1 (C0 -> J0 (C1,C0)) -> C0 -> J0 (C1,C0)
   step (W1 d c') ds                = (d,(c',ds))
   step (R1 es)   (ConsC0 (d0,ds')) = appTriple es d0 ds'
```

Case $n = m = 1$:

```
compC1 :: C1 -> C1 -> C1
compC1 c1 c2 = coitC1 costep (c1,c2)  where

   costep :: (C1,C1) -> J1 (C1,C1)
   costep (ConsC1 x1,c2) = aux x1 c2

   aux :: J1 C1 -> C1 -> J1(C1,C1)
   aux = itJ1 step

   step :: K1 C1 (C1 -> J1 (C1,C1)) -> C1 -> J1 (C1,C1)
   step (W1 d1 c1') c2          = ConsJ1 (W1 d1 (c1',c2))
   step (R1 es)     (ConsC1 x2) = subaux x2  where
```

```
subaux :: J1 C1 -> J1 (C1,C1)
subaux = itJ1 substep

substep :: K1 C1 (J1 (C1,C1)) -> J1 (C1,C1)
substep (W1 d2 c2’) = appTriple es d2 c2’
substep (R1 fs)     = ConsJ1 (R1 fs)
```

Remark. The cases shown above are also treated in [21] (without application to exact real number computation). Whereas in [21] the program was ‘guessed’ and then verified, we are able to extract the program from a proof making verification unnecessary. Of course, one could reduce Proposition 4.2 to the case $m = n = 1$, by coding n streams of single digits into one stream of n-tuples of digits. But this would lead to less efficient programs, since it would mean that in each reading step *all* inputs are read, even those that might not be needed (for example, the function $f(x, y) = x/2 + y/100$ certainly should read x more often than y).

Remark. Note that the realizability relation connecting real functions satisfying C_1 and trees in the type `C1` is much less tight than it was in the case of natural numbers (where realizability provided a one-to-one correspondence between real numbers satisfying the predicate $\mathbb{N}$ and elements of the type `Nat`). Although, by coincidence, every element of the type `C1` defines, via the program `appC` a stream transformer, this stream transformer will in general not correspond to a real function, i. e. it will not necessarily respect equality of reals represented by signed digit streams. The latter is the case only if the tree happens to realize a function f (which is of course the case if the tree was extracted from a proof of $\mathrm{C}_1(f)$). Moreover, a tree can realize $\mathrm{C}_1(f)$ for different f because the predicate C_1 says nothing about the behaviour of functions outside the interval $\mathbb{I}$.

In order to try out the programs `appC` and `compC1` one needs examples of elements of the type `C1`. Such examples will be provided in the next section.

5. Wellfounded induction and digital systems

Now we study the principle of induction along a wellfounded relation from the perspective of program extraction. As an important application we show that certain families of real functions which we call digital systems are contained in C_n. This provides a convenient tool for proving that certain functions, for example polynomials and, more generally, uniformly continuous functions on $\mathbb{I}^n$ are in C_n, and in turn allows us to extract implementations for these functions.

Wellfounded induction Let U be a set, A a subset of U and $<$ a binary relation on U. Define a monotone operator $\Phi : \mathcal{P}(U) \to \mathcal{P}(U)$ (depending on A and $<$) by

$$\Phi(X) := \{x \mid \forall y \in A(y < x \Rightarrow y \in X)\}$$

The relation $<$ is called *wellfounded* on A, $\mathrm{Wf}_A(<)$, if $A \subseteq \mu\Phi$. A set $X \subseteq U$ is called *$<$-progressive* on A, $\mathrm{Prog}_A(<, X)$, if $\Phi(X) \cap A \subseteq X$. The principle of *wellfounded induction (on A along $<$ at X)*, $\mathrm{WfInd}_A(<, X)$, is

$$\mathrm{Prog}_A(<, X) \Rightarrow A \subseteq X$$

For the purpose of program extraction let us assume that the partial order $x < y$ is defined without using disjunctions and hence has no computational content. For example, the definition could be an equation $t(x, y) = 0$ for some term $t(x, y)$. The following program realizes wellfounded induction for provably wellfounded relations (`alpha` and `beta` are the types of realizers of A and X, respectively):

```
wfrec :: ((alpha -> beta) -> alpha -> beta) -> alpha -> beta
wfrec prog = h  where  h  = prog h
```

Proposition 5.1. *If* $\mathrm{Wf}_A(<)$ *is provable, then* `wfrec` *realizes* $\mathrm{WfInd}_A(<, X)$*.*

The proof of Prop. 5.1 is beyond the scope of this introductory paper and will be given in a subsequent publication.

Remark. One can easily prove $\mathrm{Wf}_A(<) \Rightarrow \mathrm{WfInd}_A(<, X)$ from the induction principle for $\mu\Phi$ and extract a program computing a realizer of $\mathrm{WfInd}_A(<, X)$ from a realizer of $\mathrm{Wf}_A(<)$. The point is, that our realizer of $\mathrm{WfInd}_A(<, X)$ does not depend on a realizer of $\mathrm{Wf}_A(<)$.

Remark. In [32] a Dialectica Interpretation of a different form of wellfounded induction is given. There, the realizing program refers to a decision procedure for the given wellfounded relation.

Digital systems Let $(A, <)$ be a provably wellfounded relation. A *digital system* is a family $\mathcal{F} = (f_x : \mathbb{I}^n \to \mathbb{I})_{x \in A}$ such that for all $x \in A$

$$\exists d\,(f_x[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge \exists y \in A\, f_y = \mathrm{va}_d \circ f_x) \vee \exists i \bigwedge_d \exists y \in A\,(y < x \wedge f_y = f_x \circ \mathrm{av}_{i,d})$$

When convenient we identify the family $\mathcal{F}$ with the set $\{f_x \mid x \in A\}$.

Remark. The definition of a digital system makes reference to the (undecidable) equality relation between real functions. This is not a problem because, as explained in Section 2, it is not necessary for the mathematical objects and predicates to be constructively given. It is enough to be able to formulate the necessary axioms without using disjunctions (which is the case for the usual axioms for equality between functions).

Proposition 5.2. *If* $\mathcal{F}$ *is a digital system, then* $\mathcal{F} \subseteq \mathrm{C}_n$*.*

Proof. Let $\mathcal{F}$ be a digital system. We show $\mathcal{F} \subseteq \mathrm{C}_n$ by coinduction. Hence, we have to show $\mathcal{J}_n(\mathcal{F})(f_x)$ for all $x \in A$. But, looking at the definition of $\mathcal{J}_n(\mathcal{F})$ and the properties of a digital system, this follows immediately by wellfounded $<$-induction on x. □

We can extract a program from the proof of Prop. 5.2 that transforms a (realization of) a digital system into a family of trees realizing its members (case $n = 1$):

```
digitsys1 :: (alpha -> Either (SD,alpha) (Triple alpha))
             -> alpha -> C1
digitsys1 s = coitC1 (wfrec prog)  where

-- prog :: (alpha -> J1 alpha) -> alpha -> J1 alpha
   prog ih x =
     case s x of
      {Left (d,a)        -> ConsJ1 (W1 d a) ;
       Right (aN,aZ,aP) -> ConsJ1 (R1 (ih aN, ih aZ, ih aP))}
```

Example 5.3 (linear affine functions). For $\vec{u}, v \in \mathbb{Q}^{n+1}$ define $f_{\vec{u},v} \colon \mathbb{I}^n \to \mathbb{R}$ by

$$f_{\vec{u},v}(\vec{x}) := u_1 x_1 + \ldots + u_n x_n + v$$

Clearly, $f_{\vec{u},v}[\mathbb{I}^n] = [v - |\vec{u}|, v + |\vec{u}|]$ where $|\vec{u}| := |u_1| + \ldots + |u_n|$. Hence $f_{\vec{u},v}[\mathbb{I}^n] \subseteq \mathbb{I}$ iff $|\vec{u}| + |v| \leq 1$, and if $|\vec{u}| \leq 1/4$, then $f_{\vec{u},v}[\mathbb{I}^n] \subseteq \mathbb{I}_d$ for some d. Furthermore, $f_{\vec{u},v} \circ \mathrm{av}_{i,d} = f_{\vec{u}',v'}$ where $\vec{u}'$ is like $\vec{u}$ except that the i-th component is halved and $v' = v + u_i d/2$. Hence, if i was chosen such that $|u_i| \geq |\vec{u}|/n$, then $|\vec{u}'| \leq q|\vec{u}|$ where $q := 1 - 1/(2n) < 1$. Therefore, we set $A := \{\vec{u}, v \in \mathbb{Q}^{n+1} \mid |\vec{u}| + |v| \leq 1\}$ and define a wellfounded relation $<$ on A by

$$\vec{u}', v' < \vec{u}, v \quad :\Leftrightarrow \quad |\vec{u}| \geq 1/4 \wedge |\vec{u}'| \leq q|\vec{u}|$$

From the above it follows that $\mathrm{Pol}_{1,n} := (f_{\vec{u},v})_{\vec{u},v \in A}$ is a digital system. Hence $\mathrm{Pol}_{1,n} \subseteq \mathrm{C}_n$, by Proposition 5.2. Program extraction gives us a program that assigns to each tuple of rationals $\vec{u}, w \in A$ a tree representation of $f_{\vec{u},w}$. Here is the program for the case $n = 1$:

```
type Rat2 = (Rational,Rational)

linC1 :: Rat2 -> C1
linC1 = digitsys1 s  where

  s :: Rat2 -> Either (SD,Rat2) (Triple Rat2)
  s (u,v) = if abs u <= 1/4
            then let e = if v < -(1/4) then N else
                         if v > 1/4    then P
                                       else Z
                 in Left (e,(2*u,2*v-fromSD e))
            else Right (abstTriple (\d -> (u/2,u*fromSD d/2+v)))
```

In order to try this program out we introduce a utility function that applies a function $f : \mathrm{C0} \to \mathrm{C0}$ to the signed digit representation of a rational number q and computes the result with precision 2^{-n} as a rational number.

```
runC :: (C0 -> C0) -> Rational -> Integer -> Rational
runC f q n = sd2cauchy (f (cauchy2sd (const q))) (iN n)
```

Now we can compute, for example, the tree representation of the function $f(x) = \frac{1}{4}x + \frac{1}{5}$ at the signed digit representation of the point $x = \frac{1}{3}$ with an accuracy of 2^{-10} by defining

```
f :: C0 -> C0
f = appC (linC1 (1/4,1/5))
```

and evaluating the expression `runC f (1/3) 10` . The computed result, $\frac{145}{512}$, differs from the exact result, $\frac{1}{4}x + \frac{1}{5} = \frac{17}{60}$, by $\frac{1}{7680} < 2^{-10}$, as required.

Remark 5.4. In [25] it is shown that the linear affine transformations are exactly the functions that can be represented by a finite automaton. The trees computed by our program generate these automata, simply because for the computation of the tree for $f_{\vec{u},v}$ only finitely many other indices $\vec{u}', v'$ are used, and Haskell will construct the tree by connecting these indices by pointers.

Example 5.5 (iterated logistic map). With a similar proof as for the linear affine maps one can show that all polynomials of degree 2 with rational coefficients mapping $\mathbb{I}$ to $\mathbb{I}$ are in C_1. The following program can be extracted. It takes three rational numbers u, v, w and computes a tree representation of the function $f_{u,v,w}(x) := ux^2 + vx + w$, provided

$f_{u,v,w}$ maps $\mathbb{I}$ to $\mathbb{I}$. The programs **quadWrite** and **quadRead** compute the coefficients of the quadratic functions $\mathrm{va}_e \circ f_{u,v,w}$ and $f_{u,v,w} \circ \mathrm{av}_d$ while **quadTest** tests whether $f_{u,v,w}[\mathbb{I}] \subseteq \mathbb{I}_d$. Since a quadratic function may or may not have an extremal point in the interval $\mathbb{I}$ this test is more complicated than in the linear affine case.

```
type Rat3 = (Rational,Rational,Rational)

quadC1 :: Rat3 -> C1
quadC1 = digitsys1 s  where

 s :: Rat3 -> Either (SD,Rat3) (Triple Rat3)
 s uvw = case (filter (quadTest uvw) [N,Z,P]) of
                (e:_)  -> Left (e,quadWrite uvw e)
                []     -> Right (abstTriple (quadRead uvw))

 quadWrite :: Rat3 -> SD -> Rat3
 quadWrite (u,v,w) e = (2*u , 2*v , 2*w - e')
   where e' = fromSD e

 quadRead :: Rat3 -> SD -> Rat3
 quadRead (u,v,w) d = (u/4 , (u*d'+v)/2 , u*d'^2/4 + v*d'/2 + w)
   where d' = fromSD d

 quadTest :: Rat3 -> SD -> Bool
 quadTest (u,v,w) e = (e'-1)/2 <= low && high <= (e'+1)/2
   where
    e'   = fromSD e
    low  = minimum crit            -- min (f_uvw I)
    high = maximum crit            -- max (f_uvw I)
    crit = [ u+v+w, u-v+w] ++      -- [f_uvw 1, f_uvw (-1)]
           (if u == 0 then []
            else let x = -v/(2*u)          -- extremal point
                 in if -1 <= x && x <= 1
                    then [u*x^2 + v*x + w] -- f_uvw x
                    else [])
```

In particular the so-called logistic map (transformed to $\mathbb{I}$), defined by

$$f_a(x) = a(1 - x^2) - 1,$$

is in C_1 for each rational number $a \in [0, 2]$.

```
lmaC1 :: Rational -> C1
lmaC1 a = quadC1 (-a,0,a-1)
```

Exact computation of iterations of the logistic map were studied in [12] and [31]. In order to test the performance of our implementation with these maps we use a generalized exponentiation function that raises a value x to the power n (> 0) with respect to an arbitrary binary function g as “multiplication”:

```
gexp :: (alpha -> alpha -> alpha) -> alpha -> Int -> alpha
gexp g x 1 = x
gexp g x (n+1) = g (gexp g x n) x
```

Now we define a tree representing the 100-fold iteration of the logistic map f_2

```
t100 :: C1
t100 = gexp compC1 t1 100  where  t1 = lmaC1 2
```

and evaluate `runC (appC t100) 0.7 100` which means we compute $f_2^{100}(0.7)$ with a precision of 2^{-100}. The result,

$$\frac{1008550774065780194036545699607}{1267650600228229401496703205376}$$

(which is approximately 0.7956062765908836) is computed within a few seconds. Regarding efficiency, in general our experimental results compare well with those in [31] which are based on the binary signed digit representation as well. In addition, when one repeats the evaluation of the expression `(appC t100) 0.7 100` the result is computed instantly because the relevant branch of the tree `t100` has been computed before and is now memoized. The memoization effect is still noticeable if one slightly changes the iteration index or the argument `x`. Note that the function f_2^{100} is a polynomial of degree 2^{100} which oscillates about 2^{100} times in the interval $\mathbb{I}$, and the exact value of $f_2^{100}(0.7)$ is a rational number which has a (bit-)size $> 2^{100}$. Computing $f_2^{100}(0.7)$ using double precision floating point arithmetic yields the completely wrong value -0.1571454279758806 (evaluate `gexp (.) (\x-> 2*(1-x^2)-1) 100 0.7 :: Double`).

In [12] much higher iterations of logistic maps where computed (up to $n = 100,000$) by exploiting specific information about these functions to fine-tune the program. Our program, however, was extracted from completely general proofs about polynomials and composability of arbitrary u. c. functions.

An important application of digital systems is the following proof that the predicate C_n precisely captures uniform continuity. We work with the maximum norm on $\mathbb{I}^n$ and set $\mathrm{B}_\delta(\vec{p}) := \{\vec{x} \in \mathbb{I}^n \mid |\vec{x} - \vec{p}| \leq \delta\}$ for $\vec{p} \in \mathbb{I}^n$. We also set $Q := \mathbb{I} \cap \mathbb{Q}$ and let δ, ϵ range over positive rational numbers. Furthermore, we set

$$\mathrm{Box}(\delta, \epsilon, f) :\Leftrightarrow \forall \vec{p} \in Q^n \, \exists q \in Q \, (f[\mathrm{B}_\delta(\vec{p})] \subseteq \mathrm{B}_\epsilon(q))$$

It is easy to see that $f \colon \mathbb{I}^n \to \mathbb{R}$ is uniformly continuous with $f[\mathbb{I}^n] \subseteq \mathbb{I}$ iff

$$\forall \epsilon \, \exists \delta \, \mathrm{Box}(\delta, \epsilon, f) \tag{5.1}$$

Proposition 5.6. *For any function $f \colon \mathbb{I}^n \to \mathbb{R}$, $\mathrm{C}_n(f)$ iff f is uniformly continuous and $f[\mathbb{I}^n] \subseteq \mathbb{I}$.*

Proof. We have to show that $\mathrm{C}_n(f)$ holds iff (5.1) holds.

For the "if" part we use Prop. 5.2. Let A be the set of triples $(f, m, [d_1, \dots, d_k])$ such that f satisfies (5.1), $\mathrm{Box}(2^{-m}, 1/4, f)$ holds, and $d_1, \dots, d_k \in \mathrm{SD}$ with $k < n$ (hence in the case $n = 1$ the list $[d_1, \dots, d_k]$ is always empty). Define a wellfounded relation $<$ on A by

$$(f', m', [d'_1, \dots, d'_{k'}]) < (f, m, [d_1, \dots, d_k]) :\Leftrightarrow m' < m \lor (m' = m \land k' > k)$$

For $\vec{d} = [d_1, \dots, d_k]$, where $k < n$, set $\mathrm{av}_{\vec{d}} := \mathrm{av}_{1,d_1} \circ \ldots \circ \mathrm{av}_{k,d_k}$, i. p. $\mathrm{av}_{[]}$ is the identity function. We show that $\mathcal{F} := (f \circ \mathrm{av}_{\vec{d}})_{(f,m,\vec{d}) \in A}$ is a digital system (this is sufficient, because $f \circ \mathrm{av}_{[]} = f$).

Let $\alpha := (f, m, [d_1, \ldots, d_k]) \in A$.

Case $m = 0$, i. e. $\mathrm{Box}(1, 1/4, f)$. We show that the left disjunct in the definition of a digital system holds. We have $f[\mathbb{I}^n] = f[\mathrm{B}_1(\vec{0})] \subseteq \mathrm{B}_{1/4}(q)$ for some $q \in Q$. If $|q| \le 1/4$, choose $d := 0$, if $q > 1/4$, choose $d := 1$, if $q < -1/4$ choose $d := -1$. Then clearly $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$, and $g := \mathrm{va}_d \circ f$ is uniformly continuous and maps $\mathbb{I}^n$ into $\mathbb{I}$. Hence $(g, m', []) \in A$ for some m'.

Case $m > 0$. We show that the right disjunct in the definition of a digital system holds. Choose $i := k + 1$. Let $d \in \mathrm{SD}$. If $k + 1 < n$, then $\beta := (f, m, [d_1, \ldots, d_k, d]) < \alpha$ and $f \circ \mathrm{av}_{[d_1,\ldots,d_k,d]} = (f \circ \mathrm{av}_{[d_1,\ldots,d_k]}) \circ \mathrm{av}_{i,d}$. If $k + 1 = n$, then for $g := f \circ \mathrm{av}_{[d_1,\ldots,d_k,d]}$ we have $\beta := (g, m - 1, []) \in A$ because $\mathrm{av}_{[d_1,\ldots,d_k,d]}$ is a contraction with contraction factor 1/2. Clearly, $\beta < \alpha$. Furthermore, $g \circ \mathrm{av}_{[]} = g = (f \circ \mathrm{av}_{[d_1,\ldots,d_k]}) \circ \mathrm{av}_{i,d}$.

For the "only if" part we assume $\mathrm{C}_n(f)$. Set

$$E_k := \{f : \mathbb{I}^n \to \mathbb{R} \mid \exists \delta\, \mathrm{Box}(\delta, 2^{-k}, f)\}$$

For proving (5.1) it obviously suffices to show $\forall k\, (f \in E_k)$. Hence, it suffices to show $\mathrm{C}_n \subseteq E_k$ for all k. We proceed by induction on k.

Base, $k = 0$: Since $\mathrm{B}_1(0) = \mathbb{I}$, we clearly have $\mathrm{Box}(1, 2^0, f)$ for all $f \in \mathrm{C}_n$.

Step, $k \to k + 1$: Since $\mathrm{C}_n = \mathcal{J}_n(\mathrm{C}_n)$ it suffices to show $\mathcal{J}_n(\mathrm{C}_n) \subseteq E_{k+1}$. We prove this by side induction on $\mathcal{J}_n(\mathrm{C}_n)$, i. e. we show $\mathcal{K}_n(\mathrm{C}_n)(E_{k+1}) \subseteq E_{k+1}$. *Side induction base*: Assume $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$ and $\mathrm{C}_n(\mathrm{va}_d \circ f)$. By the main induction hypothesis, $\mathrm{Box}(\delta, 2^{-k}, \mathrm{va}_d \circ f)$ for some δ. Hence $\mathrm{Box}(\delta, 2^{-(k+1)}, f)$. *Side induction step*: Assume, as side induction hypothesis, $\mathrm{Box}(\delta_d, 2^{-(k+1)}, f \circ \mathrm{av}_{i,d})$ for all $d \in \mathrm{SD}$. Setting $\delta = \min\{\delta_d \mid d \in \mathrm{SD}\}$, we clearly have $\mathrm{Box}(\delta/2, 2^{-(k+1)}, f)$. □

Remark. Prop. 5.6 is mainly of theoretical value since it shows that the predicate C_n does not exclude any u. c. functions. From a practical perspective it is less useful, since, although the proof of the "if" direction computes a tree for every u. c. function f, this tree usually does not represent a very good algorithm for computing f because it follows the strategy to read *all* inputs if *some* input needs to be read (because in the proof the number m is decremented only if $k+1 = n$, i. e. all inputs have been read). Hence, for particular families of u. c. functions one should *not* use this proof, but rather design a special digital system that reads inputs only when necessary (as done in the case of the linear affine functions).

6. Integration

We prove that for functions f in C_1 the integral $\int f := \int_{-1}^{1} f = \int_{-1}^{1} f(x)\,\mathrm{d}x$ can be approximated by rational numbers, and extract from the proof a program that computes the integral with any prescribed precision. For the formal proof we do not need to define what the (Riemann- or Lebesgue-) integral is; it suffices to know that the following equations hold.

Lemma 6.1. (a) $\int f = \frac{1}{2} \int (\mathrm{va}_d \circ f) + d$

(b) $\int f = \frac{1}{2}(\int (f \circ \mathrm{av}_{-1}) + \int (f \circ \mathrm{av}_1))$.

Proof. (a) $\int (\mathrm{va}_d \circ f) = \int_{-1}^{1} (2f(x) - d)\,\mathrm{d}x = 2\int f - d\int_{-1}^{1} 1\,\mathrm{d}x = 2\int f - 2d$.

(b) By the substitution rule for integration $\int_{\mathrm{av}_d(-1)}^{\mathrm{av}_d(1)} f = \frac{1}{2}\int_{-1}^{1} (f \circ \mathrm{av}_d)$. Therefore, $\int f = \int_{-1}^{0} f + \int_{0}^{1} f = \frac{1}{2}\int_{-1}^{1} (f \circ \mathrm{av}_{-1}) + \frac{1}{2}\int_{-1}^{1} (f \circ \mathrm{av}_1)$. □

Proposition 6.2. *If* $\mathrm{C}_1(f)$*, then* $\forall k\, \exists p\, |\int f - p| \leq 2^{1-k}$*.*

Proof. We show

$$\forall k\, \forall f\, (\mathrm{C}_1(f) \Rightarrow \exists p\, |\int f - p| \leq 2^{1-k})$$

by induction on k.

$k = 0$: Since $\mathrm{C}_1(f)$ implies $f[\mathbb{I}] \subseteq \mathbb{I}$ it follows that $|\int f| \leq 2$. Hence we can take $p := 0$.

$k+1$: $\mathrm{C}_1(f)$ implies $\mathcal{J}_1(\mathrm{C}_1)(f)$. Hence it suffices to show

$$\forall f\, (\mathcal{J}_1(\mathrm{C}_1)(f) \Rightarrow \exists p\, |\int f - p| \leq 2^{-k})$$

by a side induction on $\mathcal{J}_1(\mathrm{C}_1)(f)$. If $\mathrm{C}_1(\mathrm{va}_d \circ f)$, then, by the main induction hypothesis, $|\int(\mathrm{va}_d \circ f) - p| \leq 2^{1-k}$ for some p. By Lemma 6.1 (a) it follows $|\int f - (\frac{p}{2} + d)| = \frac{1}{2}|\int(\mathrm{va}_d \circ f) - p| \leq 2^{-k}$. If $\forall d\, \exists p\, |\int(f \circ \mathrm{av}_{-1}) - p| \leq 2^{-k}$, then in particular there are p and q such that $|\int(f \circ \mathrm{av}_{-1}) - p| \leq 2^{-k}$ and $|\int(f \circ \mathrm{av}_1) - q| \leq 2^{-k}$. By Lemma 6.1 (b) it follows $|\int f - \frac{1}{2}(p+q)| = \frac{1}{2}|\int(f \circ \mathrm{av}_{-1}) + \int(f \circ \mathrm{av}_1) - (p+q)| \leq \frac{1}{2}(|\int(f \circ \mathrm{av}_{-1}) - p| + |\int(f \circ \mathrm{av}_1) - q|) \leq 2^{-k}$. □

When extracting a program from the proof of Proposition 6.2 we may treat the equations of Lemma 6.1 as axioms. The proof of Lemma 6.1 is completely irrelevant for the extracted program and was given only to convince us of the truth of the equations. Here is the program extracted from the proof of Proposition 6.2:

```
integral :: C1 -> Nat -> Rational
integral c n = aux n c  where

  aux Zero c = 0
  aux (Succ n) (ConsC1 x) = itJ1 step x  where

   step :: K1 C1 Rational -> Rational
   step (W1 d c')      = aux n c'/2 + fromSD d
   step (R1 (eN,_,eP)) = (eN + eP)/2
```

We can try it out by evaluating, for example, `integral (lmaC1 1.5) (iN 10)`.

An interesting aspect of our integration program is the fact that it "adapts" automatically to the shape of the function. For example, if we integrate a smoother function, e.g. by changing above the index $a = 1.5$ to, say, 0.1, then we can increase the precision from 2^{-10} to 2^{-20} and observe about the same computation time.

Remark. In [35] an algorithm for exact integration is given which is based on the equations of Lemma 6.1 as well and which uses ideas from [5] on a sequential implementation of the "fan functional", but where the function to be integrated is given as a continuous function on signed digit streams. Unsurprisingly, our integration program is simpler and more efficient because in our case the integrand is given as a tree containing explicit information about the modulus of uniform continuity. In general, of course, our program is still exponential in the precision which is in accordance with general results on the exponential nature of integration [24].

7. Conclusion and further work

We presented a method for extracting from coinductive proofs tree-like data structures coding exact lazy algorithms for real functions. The extraction method is based on a variant of modified realizability that strictly separates the (abstract) mathematical model the proof is about from the data types the extracted program is dealing with. The latter are determined solely by the propositional structure of formulas and proofs. This has the advantage that the abstract mathematical structures do not need to be 'constructivized'. In addition, formulas not containing disjunctions are computationally meaningless and can therefore be taken as axioms as long as they are true. This enormously reduces the burden of formalization and turns - in our opinion - program extraction into a realistic method for the development of nontrivial certified algorithms. In particular, the very short proof and extracted program for the definite integral demonstrates that our method does not become unwieldy when applied to less trivial problems.

Up to and including Sect. 3 the proof formalization and program extraction has been carried out in the Coq proof assistant. The formalization in Coq of proofs involving nested inductive/coinductive predicates such as C_n causes problems because Coq's guardedness checker does not recognize such proofs as correct. In order to circumvent these problems we are currently adapting the existing implementation of program extraction in the Minlog proof system [4] to our setting. However, we would like to stress that program extraction from proofs has turned out to be a very reliable and useful methodology for obtaining certified programs, even if the extraction is done with pen and paper and not supported by a proof assistant. We also plan to extend this work to more general situations where the interval $\mathbb{I}$ and the maps av_d are replaced by an arbitrary bounded metric space with a system of contractions (see [34] for related work), or even to the non-metric case (for example higher types). These extensions will facilitate the extraction of efficient programs for e.g. analytic functions, parametrised integrals, and set-valued functions.

Although our extracted programs perform reasonably well, we do not claim to be able to compete with existing specialized software for exact real number computation (e.g. [29, 27]) regarding efficiency. Our aim is rather to provide a practical methodology for producing correct and verified software and combining existing fully specified correct (and trusted) software components. For example, existing efficient exact implementations of certain real functions could be formally represented in our logical system as constants which are axiomatized by their given specification and realized by the existing implementation. In future work we plan to apply program extraction also to other areas, for example, monadic parsing.

Acknowledgements

I would like to thank the anonymous referees for their constructive criticism and valuable suggestions that led to several improvements of the paper.

References

[1] Abramsky, S., Jung, A.: Domain theory. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science, Volume 3. Clarendon Press (1994) 1–168

[2] Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theoretical Computer Science **333** (2005) 3–66

[3] Amadio, R., Bruce K., Longo, G.: The finitary projection model for second order lambda calculus and the solutions to higher order domain equations. In Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (1986) 122-130.

[4] Benl, H., Berger, U., Schwichtenberg, H., Seisenberger, M., Zuber, W.: Proof theory at work: Program development in the Minlog system. In Bibel, W., Schmitt, P., eds.: Automated Deduction – A Basis for Applications. Volume II of Applied Logic Series. Kluwer, Dordrecht (1998) 41–71

[5] Berger U.: Total Sets and Objects in Domain Theory. Annals of Pure and Applied Logic **60** (1993) 91–117

[6] Berger, U.: From coinductive proofs to exact real arithmetic. In Grädel, E. and Kahle, R. eds.: Computer Science Logic. Volume 5771 of Lecture Notes in Computer Science., Springer (2009) 132–146

[7] Berger, U.: Realisability and Adequacy for (Co)induction. In Bauer, A., Hertling P., Ko, K-I. eds.: 6th Int'l Conf. on Computability and Complexity in Analysis (Ljubljana). To appear: Journal of Universal Computer Science. Preliminary version available from Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany. `http://drops.dagstuhl.de/opus/volltexte/2009/2258`, 2009.

[8] Berger, U., Hou, T.: Coinduction for exact real number computation. Theory of Computing Systems **43** (2009) 394–409

[9] Berger, U., Seisenberger, M.: Applications of inductive definitions and choice principles to program synthesis. In Crosilla, L., Schuster, P., eds.: From Sets and Types to Topology and Analysis Towards practicable foundations for constructive mathematics. Volume 48 of Oxford Logic Guides. Oxford University Press (2005) 137–148

[10] Berger, U., Seisenberger, M.: Proofs, programs, processes. In Ferreira, F., Löwe, B., Mayordomo, E. and Mendes Gomes, L., eds.: Computability in Europe. Volume 6158 of Lecture Notes in Computer Science, Springer (2010) 39–48

[11] Bertot, Y.: Affine functions and series with co-inductive real numbers. Mathematical Structures in Computer Science **17** (2007) 37–63

[12] Blanck, J.: Efficient exact computation of iterated maps. Journal of Logic and Algebraic Programming **64** (2005) 41–59

[13] Bradfield, J., Stirling, C.: Modal mu-calculi. In Blackburn, P., van Benthem, J., Wolter, F., eds.: Handbook of Modal Logic. Volume 3 of Studies in Logic and Practical Reasoning. Elsevier (2007) 721–756

[14] Buchholz, W., Feferman, F., Pohlers, W., Sieg, W.: Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof–Theoretical Studies. Volume 897 of Lecture Notes in Mathematics. Springer, Berlin (1981)

[15] Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation **204** (2006) 437–468

[16] Ciaffaglione, A., Di Gianantonio, P.: A certified, corecursive implementation of exact real numbers. Theoretical Computer Science **351** (2006) 39–51

[17] Edalat, A., Heckmann, R.: Computing with real numbers: I. The LFT approach to real number computation; II. A domain framework for computational geometry. In Barthe, G., Dybjer, P., Pinto, L., Saraiva, J., eds.: Applied Semantics - Lecture Notes from the International Summer School, Caminha, Portugal. Springer (2002) 193–267

[18] Escardo, M.H., Marcial-Romero, J.R.: Semantics of a sequential language for exact real-number computation. Theoretical Computer Science **379** (2007) 120–141

[19] Geuvers, H., Niqui, M., Spitters, B., Wiedijk, F.: Constructive analysis, types and exact real numbers. Mathematical Structures in Computer Science **17** (2007) 3–36

[20] Gierz, G., Hofmann, K., Keimel, K., Lawson, J., Mislove, M., Scott, D.: Continuous Lattices and Domains. Volume 93 of Encyclopedia of Mathematics and its Applications. Cambridge University Press (2003)

[21] Ghani, N., Hancock, P., Pattinson, D.: Representations of Stream Processors Using Nested Fixed Points. Logical Methods in Computer Science **5** (2009)

[22] Ghani, N., Hancock, P., Pattinson, D.: Continuous functions on final coalgebras. Electr. Notes in Theoret. Comp. Sci. **249, 8** (2009) 3–18

[23] Hancock, P., Setzer, A.: Guarded induction and weakly final coalgebras in dependent type theory. In Crosilla, L., Schuster, P., eds.: From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics, Oxford, Clarendon Press (2005) 115 – 134

[24] Ko, K-I.: Computational Complexity of Real Functions. Birkhauser Boston, Boston, MA (1991)

[25] Konečný, M.: Real functions incrementally computable by finite automata. Theoretical Computer Science **315** (2004) 109–133

[26] Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. Constructivity in Mathematics (1959) 101–128

[27] Lambov, B.: RealLib: An Efficient Implementation of Exact Real Arithmetic. Mathematical Structures in Computer Science **1** (2007) 81-98

[28] Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14** (1990) 255–279

[29] Müller, N.: The iRRAM: Exact Arithmetic in C++. In Blanck, J. and Brattka, V. and Hertling, P. eds.: Computability and Complexity in Analysis, (CCA'2000). Volume 2064 of Lecture Notes in Computer Science., Springer (2001) 22–252

[30] Niqui, M.: Coinductive formal reasoning in exact real arithmetic. Logical Methods in Computer Science **4** (2008) 1–40

[31] Plume, D.: A Calculator for Exact Real Number Computation. PhD thesis, University of Edinburgh (1998)

[32] Schwichtenberg, H.: Dialectica Interpretation of Well-Founded Induction. Mathematical Logic Quarterly **54** (2008) 229–239

[33] Schwichtenberg, H.: Realizability interpretation of proofs in constructive analysis. Theory of Computing Systems **43** (2008) 583–602

[34] Scriven, A.: A functional algorithm for exact real integration with invariant measures. Electronic Notes in Theoretical Computer Science **218** (2008) 337–353

[35] Simpson, A.: Lazy Functional Algorithms for Exact Real Functionals. In: Mathematical Foundations of Computer Science. Volume 1450 of Lecture Notes in Computer Science, Springer (1998) 456–464

[36] Tatsuta, M.: Realizability of monotone coinductive definitions and its application to program synthesis. In Parikh, R., ed.: Mathematics of Program Construction. Volume 1422 of Lecture Notes in Mathematics, Springer (1998) 338–364

[37] Troelstra, A.: Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Volume 344 of Lecture Notes in Mathematics, Springer (1973)